Dynamic Slicing of Object-Oriented Programs.

Thesis submitted in partial fulfillment
of the requirements for the degree
Of
Bachelor of Technology
In
Computer Science and Engineering

By

Jaya Teja Gompa

Roll No: 110cs0197

Under the Guidance of

Prof. D. P. Mohapatra

May, 2014

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, 769008

Certificate

This is to certify that the project entitled "Dynamic Slicing of Object-oriented Programs", submitted by Jaya Teja Gompa, B.TECH student in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India, in the partial fulfillment for the award of the degree of Bachelor of Technology, is a record of an original research work carried out by him under our supervision and guidance. The thesis fulfills all requirements as per the regulations of this Institute and in our opinion has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. D. P. Mohapatra
Department of Computer Science and Engineering
National Institute of Technology Rourkela
India – 769008

Acknowledgement

On the submission of my Thesis report, I would like to extend my gratitude and sincere thanks to my supervisor Dr. D.P. Mohapatra, for his constant motivation and support during the course of our work in the last one year. I truly appreciate and value his esteemed guidance and encouragement from the beginning to the end of this thesis. He has been my source of inspiration throughout the thesis work and without his invaluable advice and assistance it would not have been possible for me to complete this thesis.

I would also like to give our most sincere thanks to Mr. Subhrakanta Panda, Ph.D scholar for his guidance throughout the thesis and providing me with all the resources required to carry out the thesis.

Jaya Teja Gompa

Abstract

Software maintenance activity is one of the most important part of software development cycle. Certain regions of a program cause more damage than other regions resulting in errors, if they contain bugs. So, it is important to debug and find those areas. We use slicing criteria to obtain a static backward slice of a program to find these areas.

An intermediate graphical representation is obtained for an input source program such as the Program Dependence Graph, the Class Dependence Graph and the System Dependence Graph. Slicing is performed on the System Dependence Graph using a two pass graph reachability algorithm proposed by Horwitz[3], and a static backward slice is obtained. After obtaining static slice, dynamic slice is calculated for the given input variable using an algorithm where in a statement, a set of variables and the input values for these variables are taken as input and a dynamic slice is obtained.

TABLE OF CONTENTS

1 Introduction 09

1.1 Motivation 10

1.2 Objective 10

1.3 Organization 10

2 Fundamental Concepts 12

2.1 Intermediate Graphs 12

2.1.1 Control flow graph(CFG) 12

2.1.2 Data dependence graph (DDG) 13

2.1.3 Control Dependence graph(CDG) 14

2.1.4 Program Dependence Graph(PDG) 14

2.1.5 System Dependence Graph(SDG) 15

2.1.6 Class Dependence Graph(ClDG) 16

2.2 Slicing 17

2.2.1 Input Criterion 17

2.2.2 Types 17

Based on input 17

Based on direction 18

3 Dynamic Slicing of object-oriented programs 19

3.1 Intermediate graph 19

3.1.1 Steps for constructing ClDG 20

3.1.2 Steps for constructing SDG 20

3.2 Removing Redundant Edges 20

3.3 Computation of Static Slice 36

3.4 Computation of Dynamic Slice 37

5 Implementation and results 38
5.1 Tools used 38
5.1.1 MyEclipse 38
5.1.2 Graphviz 38
5.2 Data Structures used 39
5.3 Implementation Details 41
5.3.1 PDG Graph 41
5.3.2 ClDG Graph 42
5.3.3 Removing Redundancy 44
5.3.4 Static Slice Computation 44
5.3.5 Dynamic Slice Computation 45
5.4 Result of the Dynamic Slicing algorithm 46

6 Conclusion and future work 48
6.1 Conclusion 48

References

Chapter 1

Introduction

The need for program slicing arises from the need of finding errors in the program which may effect the entire software. Many softwares have evolved in the market for these purposes. It is more preferred nowadays due to time and cost issues. Additionally, the center of building software has seen an emotional float from utilizing customary procedural strategies to protest arranged methods. Item arranged strategy, doubtlessly modularizes the system, however in the meantime, it is exceptionally perplexing and troublesome to debug and test for mistakes.

Different strategies have been produced to test virtual products for finding bugs. These strategies apply diverse methodologies to software testing which utilize different intermediate representation like SDG, ClDG etc. to represent the relations between statements in the program.

Slicing is used in software testing, regression testing and has many other applications in software maintenance activities. Static slicing is a process of selecting the statements where in all the variables change whereas dynamic slicing is selecting all the statements that change when a particular variable is taken.

1.1 Motivation:

Many literatures and procedures are proposed to compute static slices whereas very few methods are proposed to compute dynamic slice of object-oriented programs. Communication dependencies come into the picture along with object-oriented features like abstraction, polymorphism, classes etc when dealing with object-oriented programs compared to that of sequential programs.

1.2 Objective:

Our goal is to build the intermediate graph of an example object-oriented program and obtain static slice of that program and compute dynamic slice for a particular execution. Before slicing, we remove redundant edges of the graphical representation to reduce the run-time.

1.3 Organization:

The project is organized as follows:

Chapter 2

All the ground work required for the project is mentioned in this section ie; intermediate graphs, types of slicing and their differences are explained with examples.

Chapter 3

All the related work is mentioned in this section.

Chapter 4

Here, we talk about how each step of our objective is obtained and what algorithms are used to obtain them.

Chapter 5

In this section we give an overview about the tools used in different phases of slicing present the execution subtle elements of our venture to obtain graph and for coding purpose. lastly we talk about the effects.

Chapter 6

We conclude here from the results discussed in the above section.

Chapter 2

Fundamental Concepts

Here, we examine the fundamental ideas and wordings co-partnered to our work .

2.1 Intermediate Graphs

Here, we study about how to construct intermediate graphs required for slicing from a program.

2.1.1 Control Flow Graph (CFG)

It is graph with an entry and exit nodes called “START” and “STOP”. All other nodes in between are connected with edges directed in a direction to show the control flow in the program. Each statement is a node in the program.

```
x=10;
count=5;
while(count>0){
        if(x<20)
                inc(x);
        count=count-1;
}
```

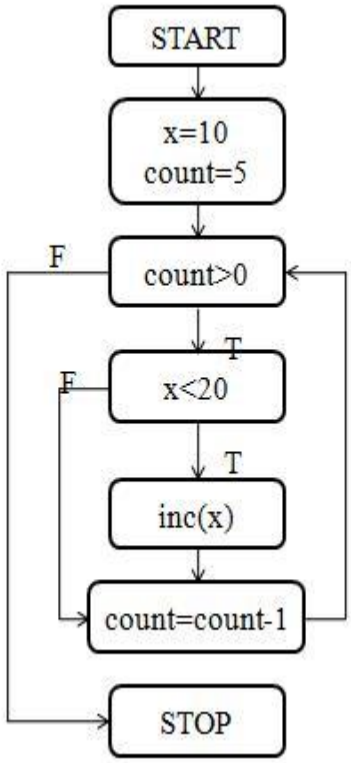


Figure 2.1: CFG

2.1.2 Data Dependence Graph (DDG)

A Data dependency edge is said to be existent if it follows the following rules:

Suppose, Consider two nodes A and B and a variable X

(i) Variable X is initiated at A

(ii) X is being used in a computing at node B

(iii) X should not defined in between and the control flow is allowed in between from A to B.

A is said to the reaching definition of B if B is data dependant on A. an example is shown showing the reaching definition of 6 and 7. Reaching definitions are calculated based in statement labels from these sets:

Def-set(S defined at), Gen-set(S generated at), Kill-set (S killed at), in-set(statement S), out-set(leaving S) etc

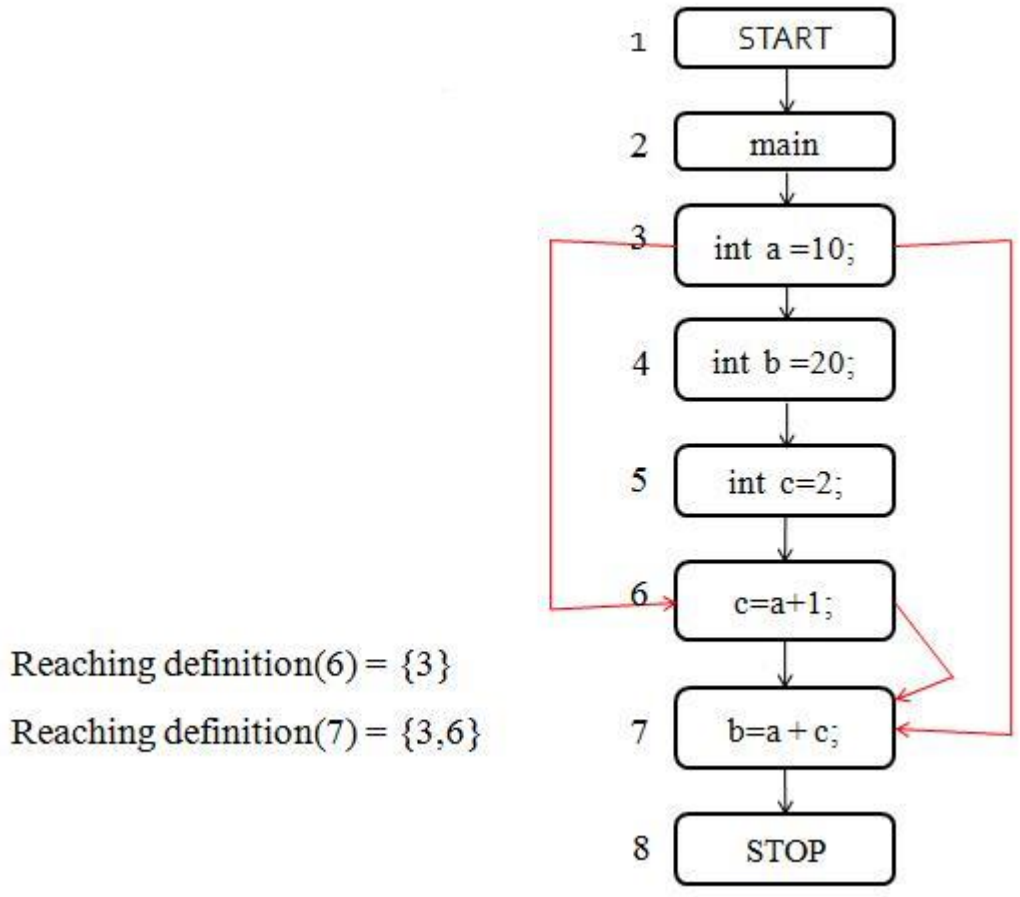


Figure 2.2: DDG

2.1.3 Control Dependence Graph (CDG)

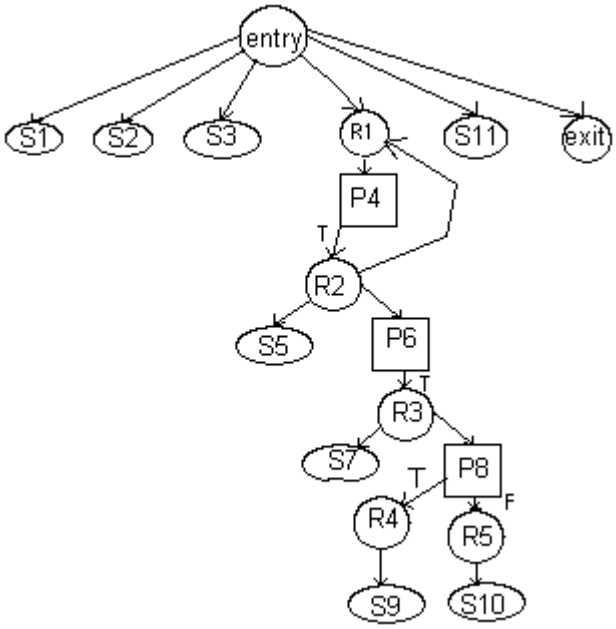
CDG summarizes the control conditions necessary for a statement to execute if a statement has executed.

A control dependence graph contains several types of nodes:

Statement nodes - represent simple statements, are shown as ellipses in the figure.

Predicate nodes - from which labeled edges originate, are shown as rectangles in the figure.

Region nodes - summarize the control-dependencies for statements in the program, are shown as circles in the figure




```
S1.    read i
S2.    sum = 0
S3.    done = 0
P4.    while i <= 5 and !done do
S5.       read j
P6.       if j >= 0 then
S7.          sum = sum + j
P8.          if sum > 100 then
S9.             done = 1
             else
S10.            i = i + 1
          endif
       endwhile
S11.   print sum
```

Figure 2.3: CDG

2.1.4 Program dependence graph (PDG)

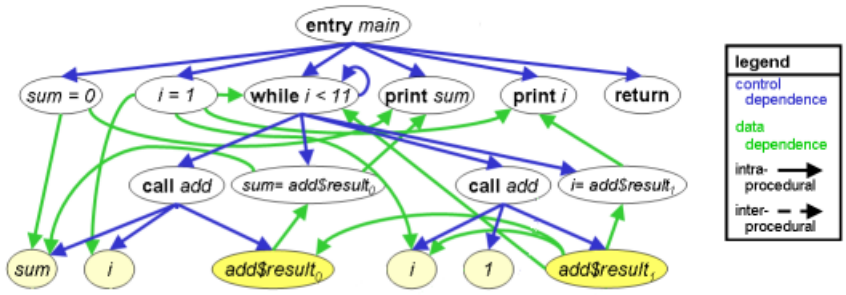
PDG is proposed by Ferrante[2] in 1987. Both data and control dependencies of a function/method are made explicit. As the DFG is updated, PDG allows incremental optimization. It is of hierarchical nature .

Disadvantage: PDG can't handle programs with multiple functions.

Fig 2.4: Sample Program(i) and it's PDG(ii)

```
void main()
{
    int sum, i;
    sum = 0;
    i = 1;
    while ( i < 11 ) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    System.out.println("sum = " + sum);
    System.out.println("i = " + i);
}
```

(i)



(ii)

In the above PDG obtained, all the control dependencies, data dependencies, intra-procedural and inter-procedural edges are marked and their flow is depicted.

2.1.5 System dependence graph (SDG)

Horwitz[3] introduced the SDG. It can handle programs with multiple proedures because it is collaboration of PDG of each procedure in the program. SDG is same as PDG for a program with single method/function. Following types of vertices are used to represent flow:

- Call site nodes represent function calls.
- Actual in-out vertices represent the calling side parameter passing.
- Formal in-out vertices represent the called procedure parameter passing.

2.1.6 Class dependence graph (ClDG)

ClDG used to speak to projects with OOPS characteristics. Each one capacity/system is spoken to in PDG and subsequently worked together with their information/control reliance edges.

Each one capacity has a capacity/technique passage vertex that addresses the area into the strategy. A CLDG furthermore holds a class section vertex that chooses the passage into the class. The class entry vertex is joined with the framework door vertex for each system in the class by a class part edge. Class section vertices and class part edges let us quickly get to the framework information when a class is united with a substitute class or schema

Formal in-out vertices are utilized to speak to parameter passing from call to capacity and the other way around.

Since the class' event variables are interested in all procedures in the class, we treat them as worldwide to schedules in the class and we incorporate formal-in and formal-out vertices for all reference variables referenced in the method. In any case, the extraordinary case to this representation for instance variable is that formal-in vertices for the event variables in the class constructor and formal-out vertices for the event variables in the class destructor are barred.

2.2 Program Slicing and it's types

Program Slicing is selection a group of statements from the initial program statements which are going to be effected if a change is made in the given input statement based on the slicing criteria.

2.2.1 criterion

Slicing criterion (S, V). where S is the statement or node number and V is the variable.It is proposed by Weiser[1]. This is for static slicing whereas for dynamic slice that particular execution for which dynamic slice to be computed also need to be mentioned.

2.2.2 Types

It is basically of two types based on input and direction.

Slicing based on input:

Static Slicing: All possible executions are considered for calculating static slice. It is calculated for an input statement wherein if the variables are made changes, what are the other statements it may change? . As all executions are acknowledged.

Dynamic slicing is computed for a particular exection only. If a variable V is changes in a statement S how does it change other statements? Since just a specific execution succession is viewed as, the predicate worth may either assess to genuine or false. Accordingly, just the real cuts are registered for a specific data.

Slicing based on direction:

- Forward slicing: All the statements which effect the statement in the slicing criteria are found out by working forward from the statement in the criteria .
- Backward Slicing: All the statements which effect the statement in the slicing criteria are found out by working backward from the statement in the criteria. We use backward slicing in our implementation.

```
s1:   main() {
s2:   int a = 1, b = 5;
s3:   a = a + b;
s4:   if (a>b) {
s5:                    a= a*b;
s6:                    b=b+2a;  }
s7:       else {
s8:                    a = a / 2;  }
s9:   }
```

Figure 2.5: A sample program

Static Slice (s4, V) ➔ {s5,s6,s8}

Whereas

Dynamic Slice (s4, V) ➔ either {s5,s6} or {s8} based the condition true or false in s4.

Chapter 3

Dynamic Slicing of object-oriented programs

Our goal is to compute dynamic slice of a program for a given slicing criteria. In order to compute dynamic slice for a particular variable, first we need to compute static slice for the statement. First, we require a graphical representation of program with appropriate edges marked. ie; ClDG, SDG etc. This representation becomes the system dependence graph for a single method. Order of constructing intermediate representation is

i) CFG

ii) DDG and CDG

iii) PDG

iv) Then the required graph depending on the purpose ie; SDG or ClDG
For concurrent programs with only thread, it becomes SDG.

After obtaining the intermediate representation, we follow these remaining steps.

- Construction of the intermediate graphical representation
- Reducing the transitive edges using Redundant Edge Removal(RER) Algorithm
- Two phase algorithm is used to compute Static slice.
- Computing the Dynamic slice from the above Static slice for a given input.

3.1 Intermediate Graphs

Algorithms used for the construction of the intermediate representation are presented here.

3.1.1 Steps for ClDG graph

Proposed by Larsen and Harrold[5] to represent object-oriented features like data hiding, inheritance and polymorphism.

Step 1: Class "ENTRY" vertex is created.

Step 2: "ENTRY" node is connected with all the method/ procedure calls.

Step 3: Each method graphs are constructed and are collaborated.

3.1.2 Steps for SDG graph

Proposed by Horowitz[3] to represent programs with more than one procedure or method.

Step 1: All methods are represented by PDGs and thus collaborated with the edges mentioned in step 2.

Step 2 : Parameter edges used for parameter passing and formal edges used for communication with call sites.

3.2 Redundant Edge Removal(RER) algorithm

Removing Redundant edges:

Given a directed graph (digraph) G= (V,E) where V is set of vertices and E is set of edges.

Ex:- (a,b) ➔ an edge incident from vertex 'a' on vertex 'b' .

An edge is redundant if it can be obtained from other ways .

RER Algorithm:

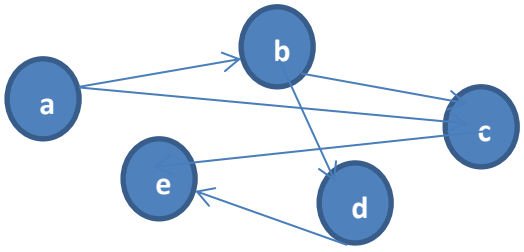
I/P: A set of Dependencies (Edges) E.

O/P: graph containing non-redundant set of edges F.

```
1.  F:= E ;
2.      For all (u,v) belongs to F do
3.          G = E – (u,v) ;
4.          S = u ;            ( S is a temporary set)
5.          For all (x,y) belongs to G do
6.              If x is subset of S then
7.                  S : = S U {y} ;
8.              End If
9.          End For
10.         If v subset of S then
11.             E := E – (u,v);
12.         End If
13.     End For
14.  F := E;            ( F is set of redundant edges)
15.  End
```

Example:-

Fig: 3.1: Sample graph



Adjacency Matrix: Adjacency Matrix of a Graph G with n vertices having no parallel edges is an n by n matrix A(G) = $[a_{ij}]$, whose elements are defined as follows:

a_{ij} = 1, if there is an edge between i^{th} and j^{th} vertices.

= 0 , otherwise

.

Adjacency matrix of the above digraph =

	a	b	c	d	e
a	0	1	1	0	0
b	0	0	1	1	0
c	0	0	0	0	1
d	0	0	0	0	1
e	0	0	0	0	0

Edge matrix em[][] (from Adjacency matrix) =

0	1	0	0
0	2	0	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

Where (em[][0],em[][1]) form edge .

Edges representation : (**0 → a, 1 → b**, **2 → c**, **3 → d**, **4 → e**)

, em[][2] = 1 , if the edge is visited.

= 0, otherwise.

And em[][3] = 1 , if the edge is redundant.

= 0, otherwise.

Let S be an array of size 'n'

S =

0	*0*	*0*	*0*	*0*

S[0] = 1, if vertex '**a**' is present in array .

= 0, otherwise.

S[1] = 1, if vertex '**b**' is present in array .

= 0, otherwise.

S[2] = 1, if vertex '**c**' is present in array .

= 0, otherwise.

S[3] = 1, if vertex '**d**' is present in array .

= 0, otherwise.

S[4] = 1, if vertex '**e**' is present in array .

= 0, otherwise.

By default , all the values are set to zero before checking redundancy for each edge.

Now, Applying the algorithm for edge set E …….

E = { (0,1), (0,2), (1,2), (1,3), (2,4), (3,4) }

1. Checking redundancy for (0,1):

G = E – (0,1) // * G = E – (u,v) *//

G = { (0,2), (1,2), (1,3), (2,4), (3,4) }

Now, set S[u] to 1 // S = u *//

ie: S[0] = 1; → **S**=

1	*0*	*0*	*0*	*0*

1st Iteration:

(i)

G ➔ (count = 0) →

x	y		
0	2	0	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

```
For all (x,y) in G ,
    If x is in S then
        Add y to S;
    End If
End for
```

For (0,2) ,

0 is in S

Set S[y] = 1;

(ie: S[2] = 1); → **S** =

1	*0*	*1*	*0*	*0*

(ii) **G ➔** (count = 1) →

x	y		
0	2	1	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

For (1,2) and (1,3)

1 is not in S .

(iii)

G ➔ (count = 1)→

x	y		
0	2	1	0
1	2	1	0
1	3	1	0
2	4	0	0
3	4	0	0

For (2,4),

2 is in S

So, S[4] = 1 →

S =

1	***0***	***1***	***0***	***1***

(iv)

G ➔ (count = 2)→

x	y		
0	2	1	0
1	2	1	0
1	3	1	0
2	4	1	0
3	4	0	0

For (3,4),

3 not in S

G ➔ (count = 2) →

x	y		
0	2	1	0
1	2	1	0
1	3	1	0
2	4	1	0
3	4	1	0

➔ *Reset em[][2] = 0, and repeat the procedure till count doesn't change (count = 0).*

2nd Iteration:

(i)

G ➔ (count = 0) →

x	y		
0	2	0	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

S =

1	*0*	*1*	*0*	*1*

For (0,2),

Both '0' and '2' are present in S.

For (1,2) and (1,3)

1 is not in S.

For (2,4),

Both '0' and '2' are present in S.

For (3,4),

1 is not in S.

G ➔ (count = 0) →

x	y		
0	2	1	0
1	2	1	0
1	3	1	0
2	4	1	0
3	4	1	0

As the count didn't change , this iterative procedure terminates.

Final value of **S** =

1	***0***	***1***	***0***	***1***

S[1] = 0, Therefore, edge (0,1) is not redundant.

2. Checking redundancy for (0,2):

G = E – (0,2) // * G = E – (u,v) *//

G = { (0,1), (1,2), (1,3), (2,4), (3,4) }

Now, set S[u] to 1 // S = u *//

ie: S[0] = 1; → **S**=

1	***0***	***0***	***0***	***0***

1st Iteration:

Count = 0 →

x	Y		
0	1	0	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

Count = 1 →

x	y		
0	1	1	0
1	2	0	0
1	3	0	0
2	4	0	0
3	4	0	0

S=

1	*1*	*0*	*0*	*0*

Count = 2 →

x	Y		
0	1	1	0
1	2	1	0
1	3	0	0
2	4	0	0
3	4	0	0

Count = 3 →

x	y		
0	1	1	0
1	2	1	0
1	3	1	0
2	4	0	0
3	4	0	0

Count = 4 →

x	y		
0	1	1	0
1	2	1	0
1	3	1	0
2	4	1	0
3	4	1	0

S =

1	*1*	*1*	*1*	*1*

S[2] = 1, Therefore, edge (0,2) is redundant.

Now, E = { (0,1), (1,2), (1,3), (2,4), (3,4) }

3. Checking redundancy for (1,2):

G = E – (1,2) // * G = E – (u,v) *//

G = { (0,1), (1,3), (2,4), (3,4) }

Now, set S[u] to 1 // S = u *//

ie: S[1] = 1; → **S**=

0	***1***	***0***	***0***	***0***

1st Iteration:

Count = 0

x	y		
0	1	0	0
1	3	0	0
2	4	0	0
3	4	0	0

→ Count = 0 →

x	y		
0	1	1	0
1	3	0	0
2	4	0	0
3	4	0	0

Count = 1 →

x	y		
0	1	1	0
1	3	1	0
2	4	0	0
3	4	0	0

Count = 1 →

x	y		
0	1	1	0
1	3	1	0
2	4	1	0
3	4	0	0

Count = 2 →

x	y		
0	1	1	0
1	3	1	0
2	4	1	0
3	4	1	0

S=

0	***1***	***0***	***1***	***1***

2nd Iteration:

Count = 0 →

x	y		
0	1	0	0
1	3	0	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	3	0	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	3	1	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	3	1	0
2	4	1	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	3	1	0
2	4	1	0
3	4	1	0

S[2] = 0, Therefore, edge (1,2) is not redundant.

4. Checking redundancy for (1,3):

G = E – (1,3) // * G = E – (u,v) *//

G = { (0,1), (1,2), (2,4), (3,4) }

Now, set S[u] to 1 // S = u *//

ie: S[1] = 1; → **S**= | 0 | 1 | 0 | 0 | 0 |

1st Iteration:

Count = 0 →

x	y		
0	1	0	0
1	2	0	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	0	0
2	4	0	0
3	4	0	0

Count = 1 →

x	y		
0	1	1	0
1	2	1	0
2	4	0	0
3	4	0	0

Count = 1 →

x	y		
0	1	1	0
1	2	1	0
2	4	1	0
3	4	0	0

Count = 2 →

x	y		
0	1	1	0
1	2	1	0
2	4	1	0
3	4	1	0

2nd Iteration:

Count = 0 →

x	y		
0	1	0	0
1	2	0	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	0	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
2	4	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
2	4	1	0
3	4	0	0

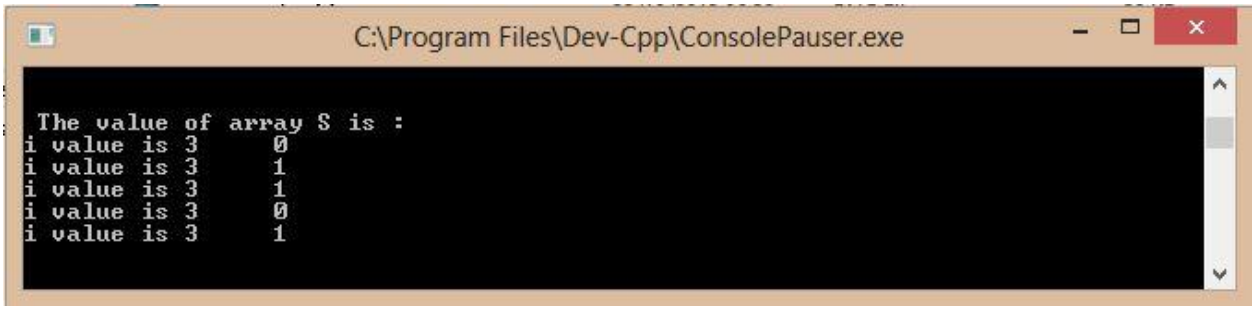
Count = 0 →

x	y		
0	1	1	0
1	2	1	0
2	4	1	0
3	4	1	0

S=

0	***1***	***1***	***0***	***1***

S[3] = 0, Therefore, edge (1,3) is not redundant.



5. Checking redundancy for (2,4):

G = E – (2,4) // * G = E – (u,v) *//

G = { (0,1), (1,2), (1,3), (3,4) }

Now, set S[u] to 1 // S = u *//

ie: S[2] = 1; → **S**=

0	*0*	*1*	*0*	*0*

1st Iteration:

Count = 0 →

x	y		
0	1	0	0
1	2	0	0
1	3	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	0	0
1	3	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
1	3	0	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
1	3	1	0
3	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
1	3	1	0
3	4	1	0

S=

0	*0*	*1*	*0*	*0*

S[4] = 0, Therefore, edge (2,4) is not redundant.

6. Checking redundancy for (3,4):

G = E – (3,4) // * G = E – (u,v) *//

G = { (0,1), (1,2), (1,3), (2,4) }

Now, set S[u] to 1 // S = u *//

ie: S[3] = 1; → **S**=

0	*0*	*0*	*1*	*0*

1st Iteration:

Count = 0 →

x	y		
0	1	0	0
1	2	0	0
1	3	0	0
2	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	0	0
1	3	0	0
2	4	0	0

Count = 0 →

x	Y		
0	1	1	0
1	2	1	0
1	3	0	0
2	4	0	0

Count = 0 →

x	y		
0	1	1	0
1	2	1	0
1	3	1	0
2	4	0	0

Count = 0 →

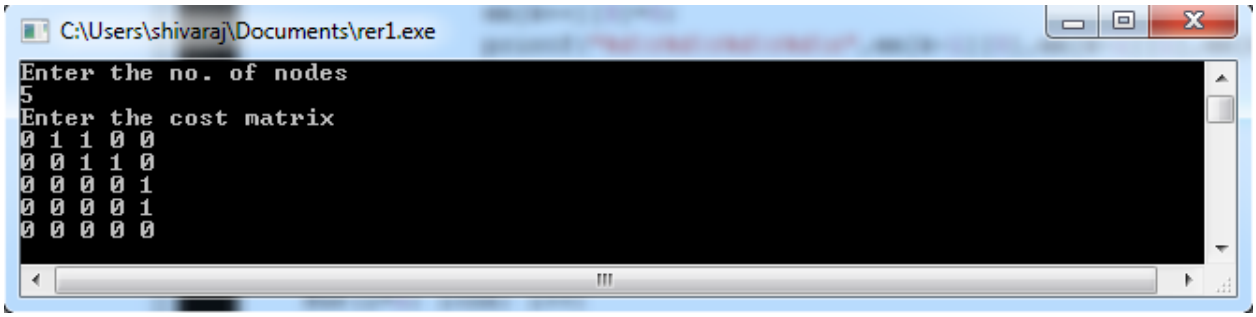
x	y		
0	1	1	0
1	2	1	0
1	3	1	0
2	4	1	0

S=

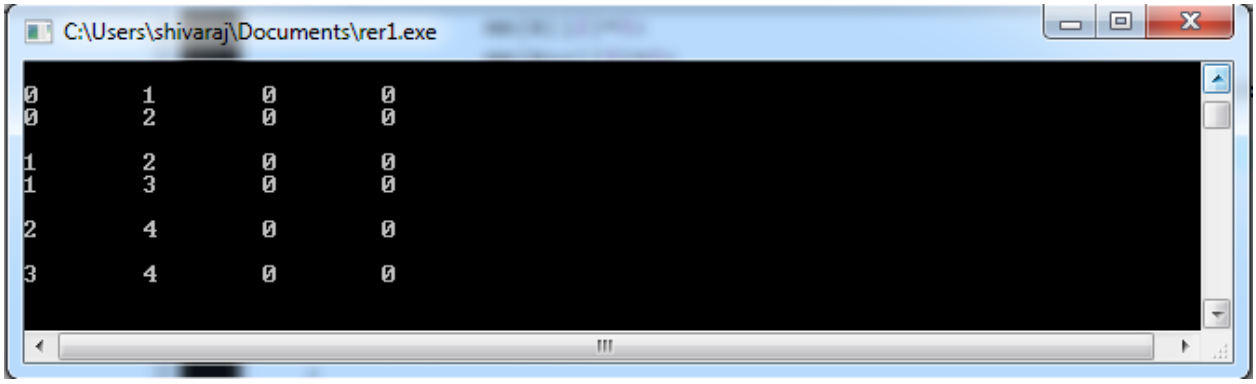
0	*0*	*0*	*1*	*0*

S[4] = 0, Therefore, edge (3,4) is not redundant.

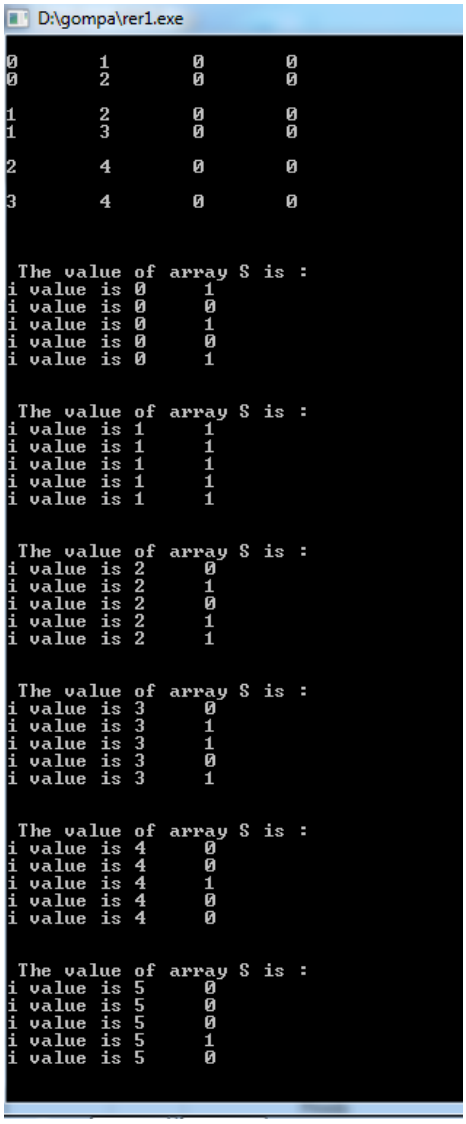
INPUT:



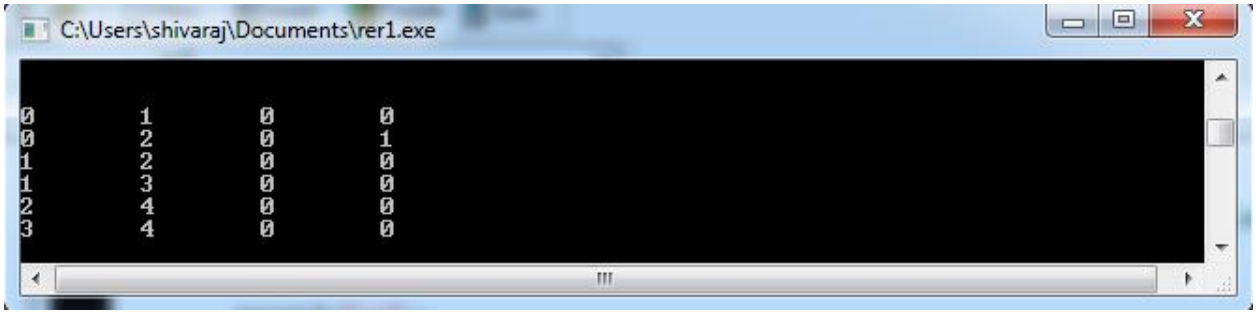
EDGE MATRIX (I/P):



ARRAY S[]:

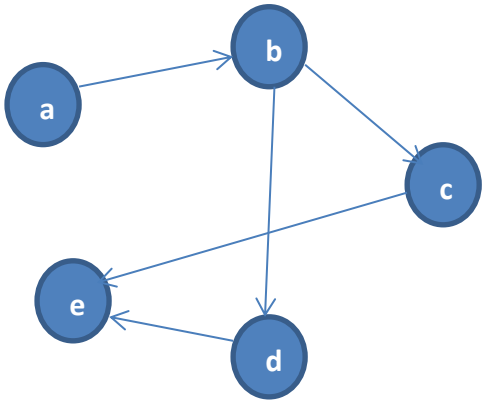


EDGE MATRIX (O/P):



OUTPUT:

RESULTANT GRAPH:



3.3 Computation of static slice.

Static Slice is computed by considering the required edges from the input node in two phases proposed by Horwitz[3]. DFS search algorithm is employed here as keeping track of the visited and unvisited nodes is required using stack.

Phase 1: In phase 1, we slice without descending into called procedures by marking reaching vertices Ie; all the definite order edges and parameter-out edges of the visited nodes.

Phase 2: In phase 2, we slice called procedures without ascending into call sites by marking all definite-order edges, parameter-in edges and the call edges of the visited nodes

A work list is maintained to keep track of the nodes and their corresponding edges. Stack is the data structure employed in this process.

Now, we obtain a set of statements which are connected with specific edges mentioned in both phases.

3.4 Computation of Dynamic Slice.

We have all the statements selected in Static Slice which effect the statement in the above step. Now a table is created for all the statements where in the variables are being effected/changed. The Advantages of using a table in runtime are no new nodes need to be created and added to the intermediate representation at run-time. No trace files are required to be maintained which saves expensive node creation and file i/o steps. Another important advantage is when the request for a slice is made , it is already available.

Implementation:

A matrix is generated taking x-coordinates as statement numbers and y-coordinates as variables V during run-time. Variables V1, V2,V3….Vn represent n variables.

Consider this example: Let the variables V1, V2, V3, V4 be x, y, z, a respectively.

	x	y	z	a
S0	0	1	1	0
S1	1	0	1	0
S2	1	1	0	0
S3	1	1	0	1

Edge 1 in (S0,y) represents that there is a change made to the variable in statement S0. Similarly, all the edges are marked and a matrix is generated.
This matrix is generated during run-time from the graphical representation. This matrix is used to compute the dynamic slice and a set of statements are obtained.

Chapter 4

Implementation and results

Here, implementation details of each of the four steps are presented with snapshots.

4.1 Tools used

These tools are used to execute the four steps of our project (both graphical representation and coding purpose):

- MyEclipse
- Graphviz

4.1.1MyEclipse

MyEclipse Enterprise Workbench is a full-featured, Enterprise-class platform and tool suite for developing software applications and systems supporting the full life-cycle of application development. Facilities and features usually found only in high-priced, Enterprise-class products are included in MyEclipse. Based on open-industry standards and the Eclipse platform, MyEclipse redefines software pricing, support and delivery release cycles by providing a complete application development environment for J2EE WEB, XML, UML and databases and the most comprehensive array of application server connectors (25 target environments) to optimize development, deployment, testing and portability.

4.1.2 Graphviz

Graphviz pictorially represent a graph. This tool in used in my project for visualization of output in a better way. The Graphviz layout programs take descriptions of graphs in a simple text language, and make diagrams in useful formats, such as images and SVG for web pages; PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks.

4.2 Implementation of Data Structures(DS)

Here in the project, Depth first search is used for traversing and calculating slices. First, Static slice is calculated from the vertices(nodes) and the different types of edges taken as input. We have used DFS approach to keep track of the nodes as well as the path. All the slices are calculated by solving a node reachability problem in the graph.

Why DFS over BFS?

Analyzing BFS and DFS, the enormous focal point of DFS is that it has much lower memory necessities than BFS, in light of the fact that its not important to store the greater part of the youngster pointers at each one level. Contingent upon the information and what you are searching for, either DFS or BFS could be worthwhile.

Case in point, given a family tree if one were searching for somebody on the tree who's still alive, then it might be sheltered to expect that individual might be on the bottom of the tree. This implies that a BFS might take quite a while to achieve that last level. A DFS, in any case, might discover the objective speedier. At the same time, if one were searching for a relative who passed on quite a while prior, then that individual might be closer to the highest point of the tree. At that point, a BFS would typically be quicker than a DFS. Along these lines, the favorable circumstances of either differ relying upon the information and what you're searching for.

Classes used are:

1) DynamicSlice
2) Graph
3) Vertex
4) StackX

4.2.1 DynamicSlice:

" DynamicSlice" is our main class where in all the vertices and edges are mentioned. All edges and vertices are given as input using an object "theGraph" in Java.

Edges -- theGraph.addEdge(a,b)

Vertex -- theGraph.addVertex(a)

4.2.2 Vertex:

This class is used to keep track of visited and unvisited adjacent nodes based on which the nodes are popped and pushed into the Stack.

4.2.3 StackX:

When a node is visited visited, it is pushed into the stack and based on visited and unvisited adjacent nodes pop() and push() operations are done . Functions used here are

push(), pop(), peek() and isEmpty()

4.2.4 Graph:

Graph is the object used here. Functions implemented here are :

i) addVertex(String)
ii) addEdge(int)
iii) displayVertex(int)
iv) dfsSearch1(int)
v) getAdjUnvisitedVertex(int)

4.3 Screenshots of implementation

Different screenshots of the implementation of the various dependence graphs have been shown taking a few examples.

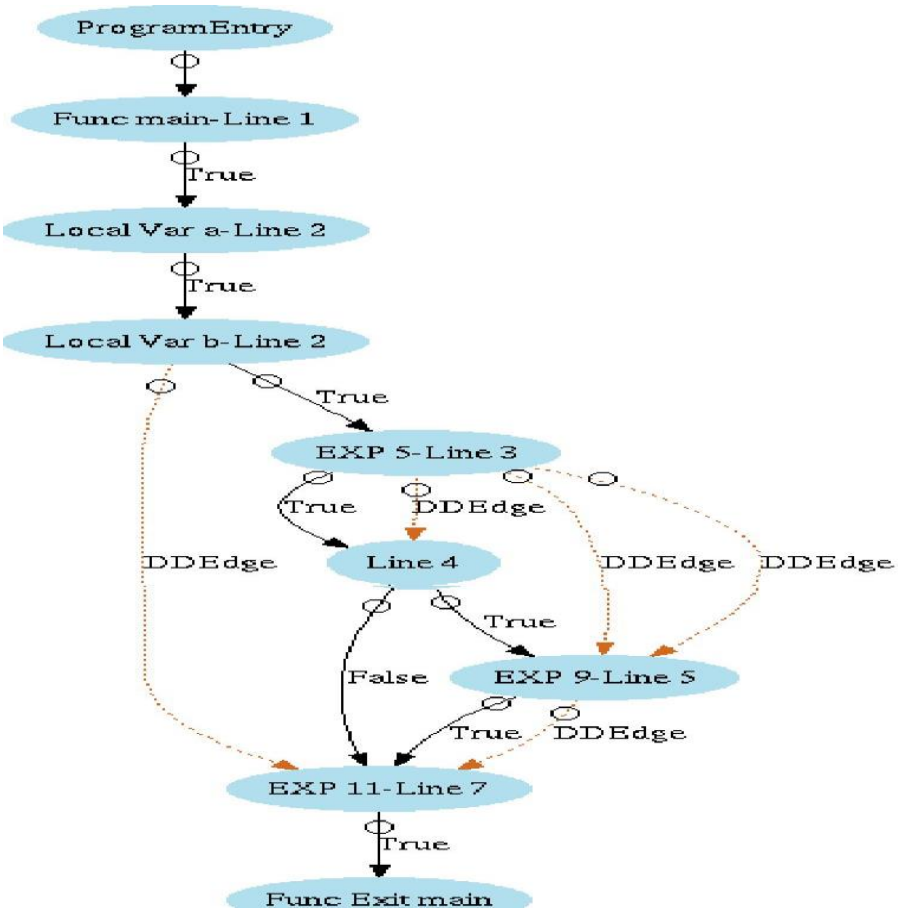
```
main() {
    int a; int b; a =10;
    if (a>5)
        b = a + a; cout << b;
    }
```

Fig 4.1: A sample program

```
main() {
        int x= 10, y=20;
        LCM(x,y);  }
int LCM(int a, int b) { int z;
  z = a + b;
     return z;
  LCM(z,x);
  }
```

Fig 4.2: A sample program

4.3.1 Implementation of CFG { Fig 4.3: CFG of sample program in Fig 4.1 }



4.3.2 Implementation of PDG

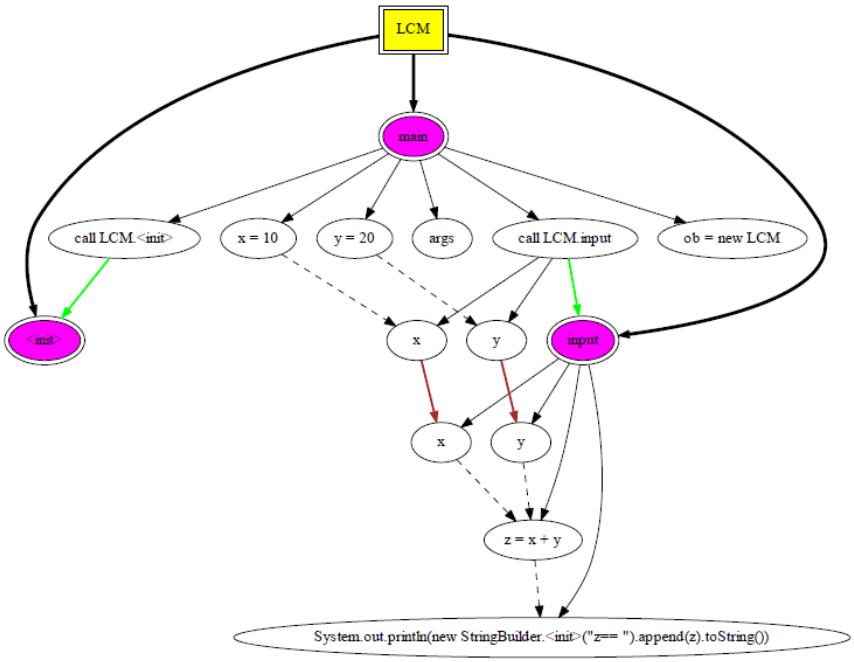


Fig 4.4: PDG of program in example 4.2

4.3.2 Implementation of ClDG

```
1:  class Elevator{
        public:
2:      Elevator(int l_top_floor)    /* inittaltzation for Elevator */
3:        { current_floor = 1;
4:          current_direction = UP;
5:          top_floor = l_top_floor; }     /* end of Elevator */
6:      virtual  ~Elevator()  { }
7:      void up()
8:        { current_direction = UP; }
9:      void down()
10:       {  current_direction = DOWN;  }
11:     int which_floor()
12:       { return current_floor;  }
13:     Direction direction()
14:       { return current_direction;  }
15:     virtual void go(int floor)   /* declaration for method go() */
16:       { if  (current_direction = UP )
17:             { while (current_floor != floor)
                      &&  (current_floor <= top_floor)
18:                 add(current_floor, 1); }
                else
19:             { while (current_floor != floor)
                      && (current_floor  > 0 )
20:                 add(current_floor, -1);  } /* end if */
          };
        private:
21:     add(int &a, const int &b)/* This method computes value of current_floor */
22:         {  a = a+b; } ;
        protected:
          int current_floor;
          Direction current_direction;
          int top_floor;
```

```
        };
23:     class AlarmElevator: public Elevator {  /* AlarmElevator is derived from  Elevato
        public:
24:       AlarmElevator(int top_floor);
25:           Elevator(top_floor)
26:               {alarm_on = 0;   }
27:       void set_alarm()
28:               {alarm_on = 1;  }
29:       void reset_alarm()
30:               {alarm_on = 0;  }
31:       void go(int floor)
32:           { if (! alarm_on)
33:               Elevator :: go(floor);
                  };
        protected:
            int alarm_on;
        };
34: main(int argc, char  **argv)  {
        Elevator *e_ptr;
35:     if (argv[1])
36:           e_ptr = new Elevator(10);
         else
37:           e_ptr = new AlarmElevator(10);
38:     e_ptr -> go(3);      /* polymorphic method call */
39:     cout << "\n currently on floor:"
              << e_ptr -> which_floor();
        } /* end of main */
```

Fig 4.5: A sample program

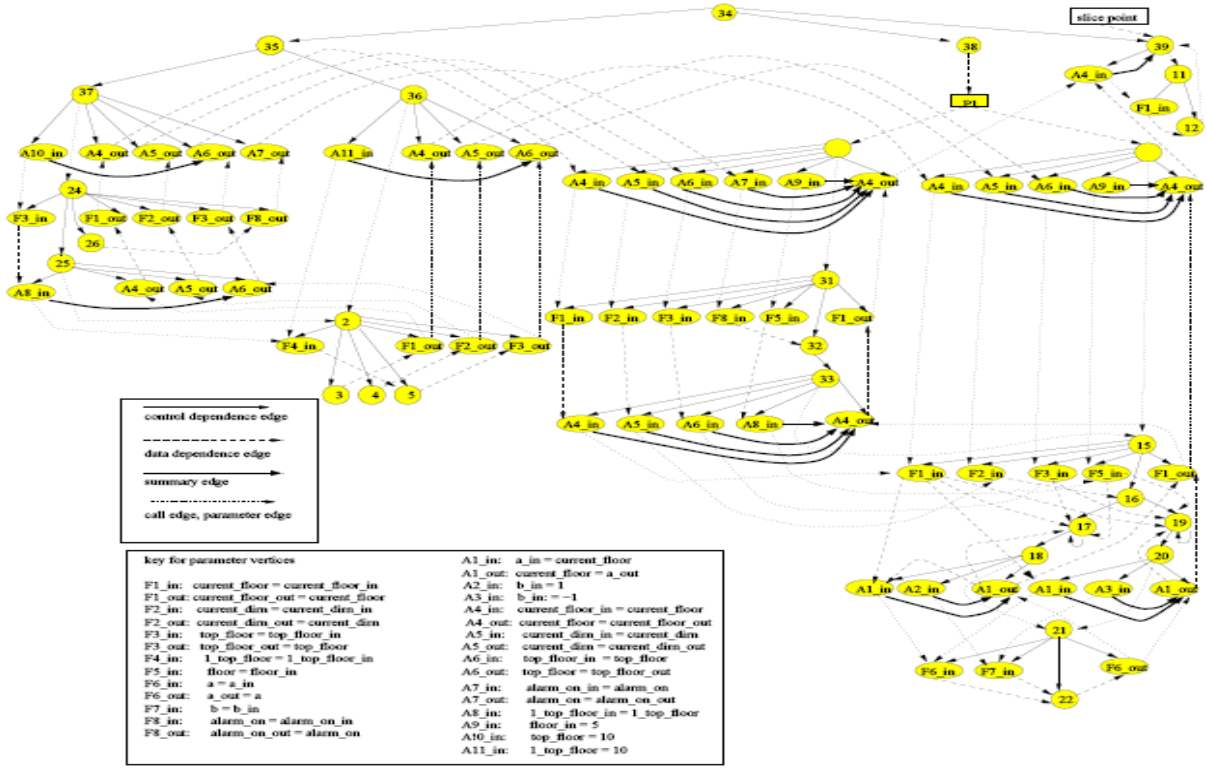


Fig 4.6: ClDG of program in Fig 4.5

4.3.3 Redundant Edge Removal :

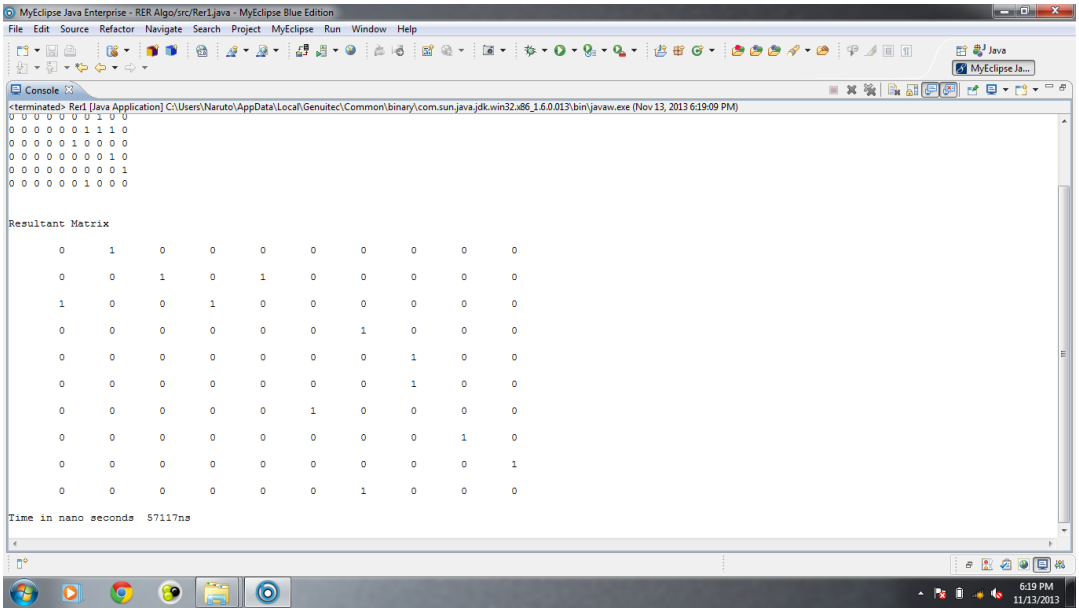


Fig 4.7: Reduced graph of Fig 4.6

4.3.4 Static Slice Computation:

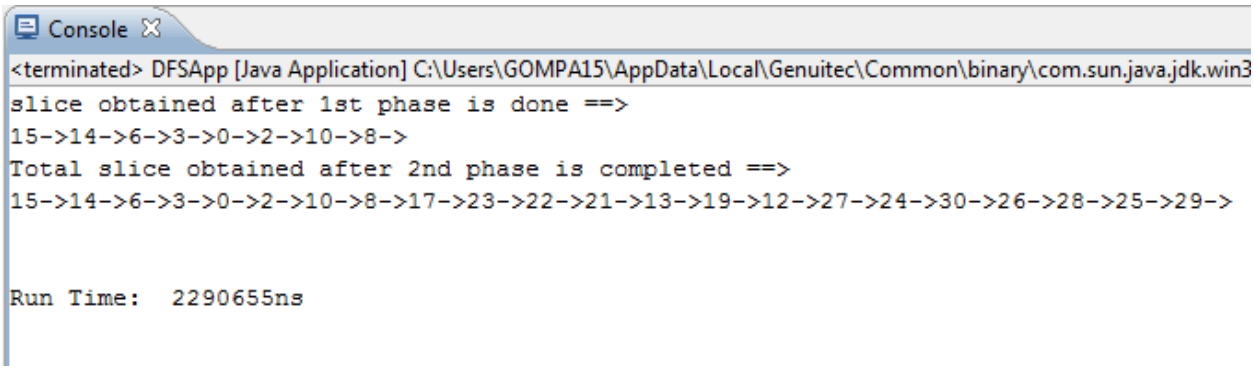


Fig 4.8: Slicing for S(15,V) of Fig 4.6

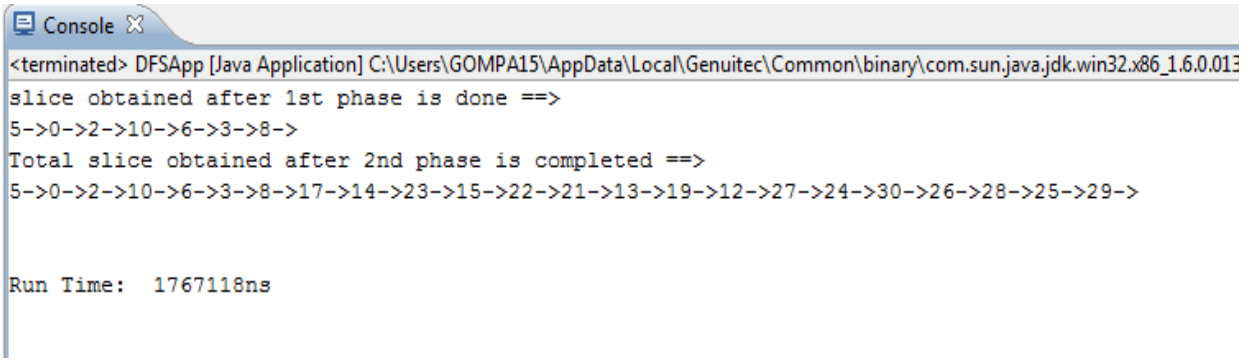


Fig 4.9: Slicing for S(5,V) for Fig 4.6

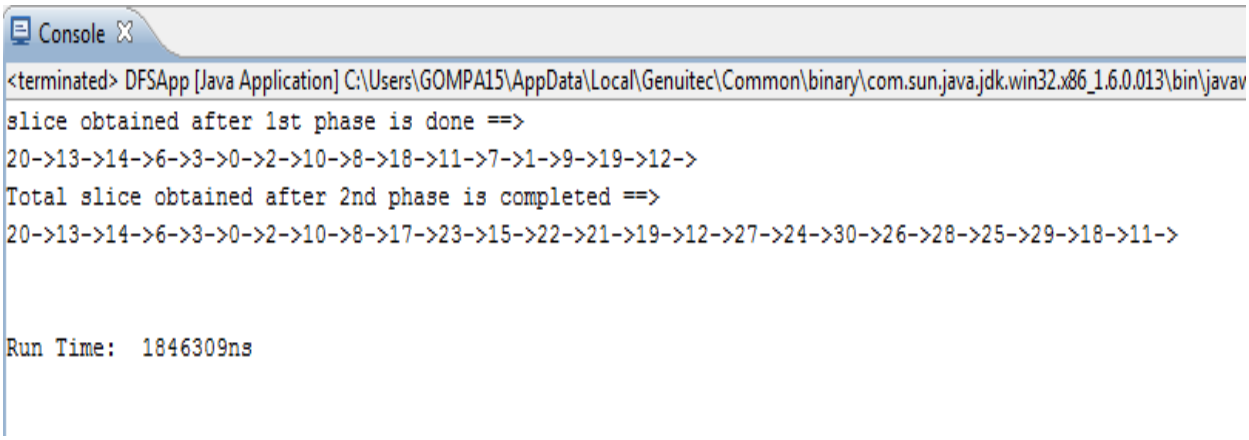


Fig 4.10: Slicing for S(20,V) for Fig 4.6

5.3.5 Dynamic Slice Computation:

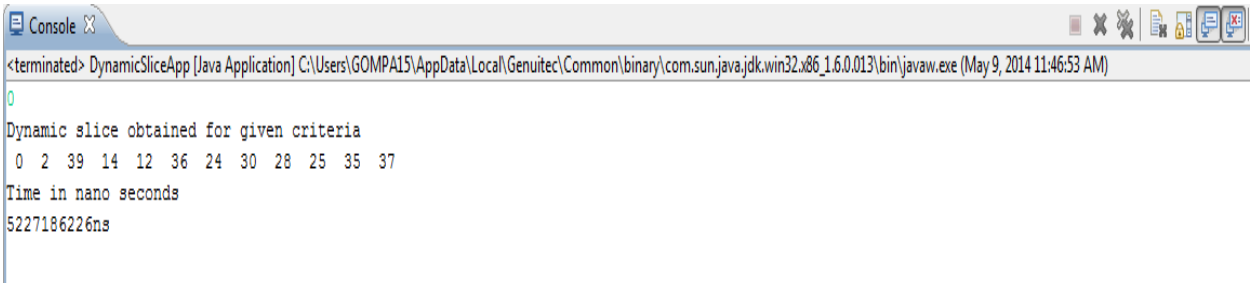


Fig 4.11: Dynamic Slice for S(5,a) of Fig 4.8

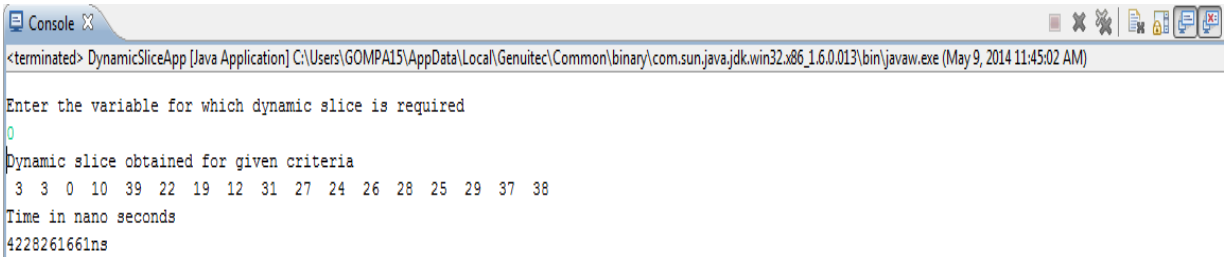


Fig 4.12: **Dynamic Slice for S(15,a) of Fig 4.9**

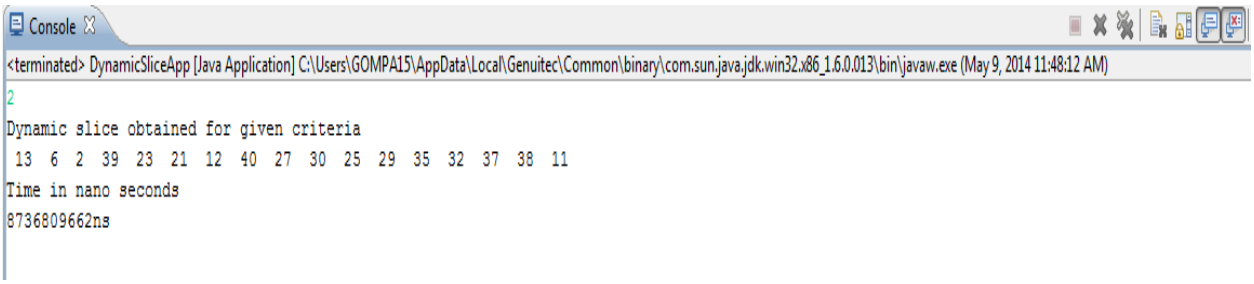


Fig 4.13: Dynamic Slice for S(20,c) of Fig 4.10

4.4 Results obtained after dynamic slicing:

S. No.	No. of Statements	Average time for obtaining dynamic slice(in µs)
1	5	2142.20
2	10	2174.75
3	20	2305.91
4	30	2443.86
5	40	2618.70

Graph:

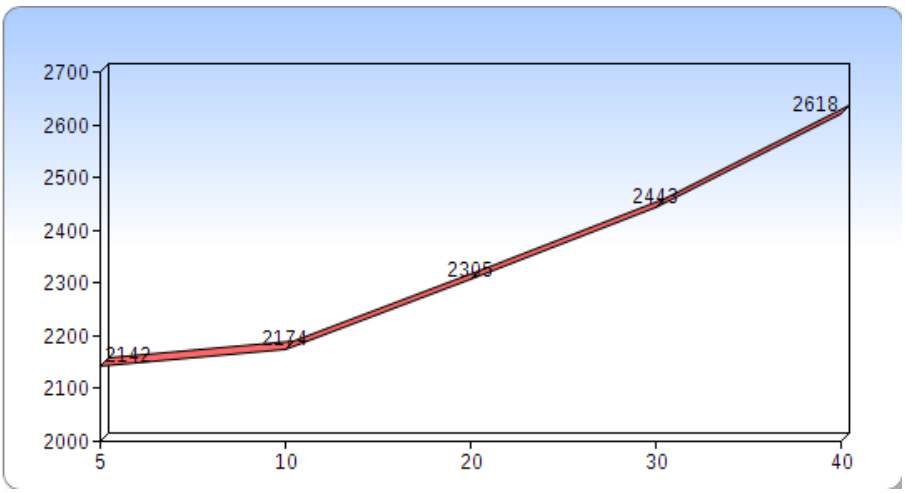


Fig 4.14: Graphical representation of results

Results:

These results depict that for programs with small number of statements (below 10), the change is average dynamic slice time has a very slight increase almost doesn't change much when compared to individual programs. Here the slight increase is significant beacause we are considering the average time.

For programs with more than 10 statements, the increase in average time is quite significant upto 30 statements

For programs with statements more than 30 statements, we can see that there is slight slant upwards. This shows that average dynamic slice increase is more than of those programs with 10-30 statements.

Graph Conclusion:

As number of statements of input program increases, average dynamic slice time increases up to a certain number of statements. After that, there is quite an increase in time.

Chapter 5

Conclusion and future work

5.1 Conclusion

I have taken a sample program and represented it's ClDG proposed by Larsen and Harold[5] as intermediate representation. I implemented the Redundant Edge Removal Algorithm to remove the redundant edges from the intermediate graph representation. Once an intermediate representation is obtained, I implemented the two-phase algorithm proposed by Larsen and Harrold[5] to compute a static backward slice for a sample input program and further computed dynamic slice for required variables. I have computed average time it takes to compute dynamic slice for programs with different number of statements and thus plotted a graph based on the obtained values. Results are thus concluded after obtaining.

5.2 Future Work:

Dynamic slicing of object-oriented programs is done but work needs to be done on Concurrent object-oriented programs, Distributed object-oriented programs and Web-based applications. With the increase in significance of these programs in today's world, there is a strong need for more research work in these areas.

References

[1] Weiser, Mark. "Program slicing." Proceedings of the 5th international conference on Software engineering. IEEE Press, 1981.

[2] Ferrante , Jeanne, Ottenstein K. J, and Joe D. Warren. " The Program Dependence graph and its use in optimization." ACM Transactions on Programming Languages and Systems (TOPLAS) 9.3 (1987): 319-349.

[3] Horwitz, Susan, Thomas Reps, and David Binkley. "Inter-procedural slicing using dependence graphs." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.1 (1990): 26-60.

[4] Krishnaswamy , Anand. "Program slicing: An application of object-oriented program dependency graphs." Clemson: Department of Computer Science, Clemson University (1994).

[5] Larsen, Loren, and Mary Jean Harrold. "Slicing object-oriented software."Software Engineering, 1996., Proceedings of the 18th International Conference on. IEEE, 1996.

[6] Nanda, Mangala Gowri, and S. Ramesh. "Slicing concurrent programs." ACM SIGSOFT Software Engineering Notes. Vol. 25. No. 5. ACM, 2000.

[7] Mohapatra, Durga Prasad, Rajib Mall, and Rajeev Kumar. "An overview of slicing techniques for object-oriented programs." Informatica (Slovenia) 30.2 (2006): 253-277.

[8] Xu B., Qian J., Zhang X., Wu Z., and Chen L., A Brief Survey of program slicing, ACM SIGSOFT Software Engineering Notes 30, Pages 1-36, February 2005.